Attorney Docket No.: 42P17527 *PATENT*

# UNITED STATES PATENT APPLICATION

For

# EFFICIENT SYSTEM MANAGEMENT SYNCHRONIZATION AND MEMORY ALLOCATION

INVENTORS:
Barnes Cooper, Grant H. Kobayashi

Prepared By:
BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

**(408) 720-8300**

"Express Mail" mailing label number: EV339922595US
Date of Deposit: October 6, 2003
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, Virginia 22313-1450
Patricia M. Richard
(Typed or printed name of person mailing paper or fee)
(Signature of person mailing paper or fee)
10/6/03 (
Date signed)

# Efficient System Management Synchronization and Memory Allocation

## FIELD

[0001] This invention relates to the field of computer systems and, in particular, to system management mode optimizations.

## BACKGROUND

[0002] Computer systems are becoming increasingly pervasive in our society including everything from small handheld electronic devices, such as personal digital data assistants and cellular phones, to application-specific electronic components, such as set-top boxes and other consumer electronics, to full mobile, desktop, and server systems. However, as systems become smaller in size and in price, the need for efficient memory allocation and system management becomes more important.

[0003] Server systems have been traditionally characterized by a significant amount of conventional memory and multiple physical processors in the same system (a multiprocessor system), wherein a physical processor refers to a single processor die or single package. The significant amount of resources available to a server system has lead to extremely inefficient allocation of memory space and wasted execution time.

[0004] Typically, there are two types of system management interrupts (SMIs) that may be generated in a system, which include hardware (asynchronous) SMIs, such as a battery being low, or software (synchronous)

SMIs, such as an operating system (OS) requesting a processor to change frequency or power levels. Usually, a hardware SMI may be handled by either processor without the knowledge of the other processor's save-state area.

[0005] However, a software generated SMI may require all the processors in a multiprocessor system to enter SMI before handling the SMI request, because handling a software SMI may require the ability to access each processor's save-state area. The process of having a plurality of processors enter system management mode before handling a SMI is commonly known as synchronization.

[0006] Current multiprocessor systems typically utilize an inefficient timeout method for synchronizing processors. For example, if an SMI is received, each processor may wait a specified amount of time before handling the SMI to ensure each processor has entered system management mode (SMM). As an illustrative example, a processor, in a multiprocessor system, may wait the amount of time it takes to execute the longest instruction before handling the SMI to ensure the other processors have entered SMM. As a result, each processor may have already entered SMI, but the system is sitting idle wasting execution time waiting for the timeout period to expire.

[0007] Furthermore, present multiprocessor systems allocate system management memory space inefficiently. Due to current addressing limitations, a typical SMM area may require at least 64kB. However, not all of this memory space is filled by SMM code and/or data. In addition, each processor is usually assigned separate and distinct 64kB SMM spaces. Therefore, each 64kB SMM space has memory space that is not be utilized, but is dedicated to an individual processor.

[0008] Nevertheless, these inefficient methods of synchronization and system management memory allocation are not limited to multiprocessor server systems. In fact, these inefficiencies may exist in other systems, such as mobile multiprocessor systems. Hyper-Threading Technology (HT) is a technology from Intel® Corporation of Santa Clara, California that enables execution of threads in parallel using a signal physical processor. HT incorporates two logical processors on one physical processor (the same die). A logical processor is an independent processor visible to the operating system (OS), capable of executing code and maintaining a unique architectural state from other processor in a system. HT is achieved by having multiple architectural states that share one set of execution resources.

[0009] Therefore, HT enables one to implement a multi(logical)processor system in a mobile platform. As shown above, inefficient memory allocation and processor synchronization exist in traditional multiprocessor systems, such as server systems. Accordingly, as multiprocessor systems begin to infiltrate the mobile realm, where resources such as memory are limited, the need for optimizations of the aforementioned inefficiencies becomes even more important.

## BRIEF DESCRIPTION OF THE DRAWINGS

[0010] The present invention is illustrated by way of example and not intended to be limited by the figures of the accompanying drawings.

[0011] Figure 1 illustrates a block diagram of a device with multiple processors that share execution resources, caches, and storage.

[0012] Figure 2a illustrates a block diagram of a system with multiple processors coupled to a storage medium.

[0013] Figure 2b illustrates a block diagram of a system with multiple processor coupled to a controller hub, which is coupled to memory

[0014] Figure 3 illustrates a block diagram of a system with a physical processor having multiple logical processors.

[0015] Figure 4 illustrates the storage medium of Figure 3 with overlapping system management memory space.

[0016] Figure 5 illustrates a portion of memory space from Figure 4, which may store representations of multiple processor's system management states.

[0017] Figure 6 illustrates a flow diagram for synchronizing a first and second processor before handling a system management interrupt.

[0018] Figure 7 depicts a flow diagram of an illustrative embodiment for synchronizing a first and second processor before handling an SMI and for updating a storage medium to reflect the new system management state.

**[0019]** Figure 8 depicts an illustrative example of a synchronization byte being used during a boot process to synchronize a first and second processor before handling a SMI.

**[0020]** Figure 9 illustrates a flow diagram for efficiently assigning system management memory space to a first and second processor.

**[0021]** Figure 10 illustrates the flow diagram of Figure 9, wherein the overlapping portion of assigned memory spaces stores system management states of at least one processor.

## DETAILED DESCRIPTION

[0022] In the following description, numerous specific details are set forth such as examples of specific memory addresses, memory sizes, and component configurations in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that these specific details need not be employed to practice the present invention. In other instances, well known components or methods, such as routine boot-up steps (e.g. power on self-test (POST)), specific system management mode (SMM) implementation, and specific system management interrupt handler code have not been described in detail in order to avoid unnecessarily obscuring the present invention.

[0023] The method and apparatus described herein are for optimization of multiprocessor synchronization and allocation of system management memory space. Synchronization of processors may occur at any time before entering a different processor mode. For example, when a plurality of processors in a multiprocessor system receive a SMI, they may synchronize before handling the SMI.

[0024] It is readily apparent to one skilled in the art, that the method disclosed for synchronizing multiple processors and efficiently allocation system management space between multiple processors may be applicable to any level computer system (personal digital assistants, mobile platforms, desktop platforms, and server platforms), as well as any number of processors. For example, a multiprocessor system with four or more processor may use this method to synchronize all four processors before entering a system management mode (SMM). As any plurality of processors would necessarily include two processors, only two processor synchronization will be discussed herein, so as to not obscure the invention with a more complex system.

[0025] **Figures 1-3** depict illustrative examples of some hardware that may embody the methods described herein. As stated above, the methods described herein may be used in any multiprocessor system; therefore, the methods will only be described in detail with reference to Figure 3, as not to obscure the invention with repetition.

[0026] **Figure 1** illustrates a block diagram of a device 105 with multiple logical processors. A physical processor refers to a physical processor die or a single package. A logical processor is an independent processor visible to the operating system (OS), capable of executing code and maintaining a unique architectural state from other processors in the system. Hyper-Threading Technology (HT) is a technology from Intel® Corporation of Santa Clara, California that enables execution of threads in parallel using a signal physical processor. HT includes two logical processors on one physical processor and is achieved by duplicating the architectural state, with each architecture state sharing one set of processor execution resources.

**[0027]** Device 105 may include a first processor 110 and a second processor 115. Device 105 may be a physical processor. Device 105 may also be an embedded system, a single physical processor, or any other device having at least two processors. Processors 110 and 115 may be logical processors. For example, device 105 may include architecture state registers 120 and 125 that each holds a unique architecture state. It is readily apparent that device 105 may include more than two logical processors that each have an architecture state register associated with it to hold a separate architecture state. Processors 110 and 115 share the same execution resources 130, caches 135, and storage medium 140.

**[0028]** Storage medium 140 may be any style of storage where data may be stored. For example, storage medium 140 may be registers to store information. Storage medium 140 may also be another level of cache 135. Storage medium 140 may also be a form of system memory placed on device 105.

**[0029]** Turning to **Figure 2a**, an illustrative example of a system with multiple processors is depicted. The system may include first processor 205 and second processor 210. Processor 205 and 210 may be physical processors, wherein each processor is on a separate die or separate package. The system may also include interconnect 215 to couple processors 205 and 210 to either storage medium 220 in **Figure 2a** or controller hub 230 in **Figure 2b**. As shown in **Figure 2b**, controller hub 230 may also be coupled to storage medium 220 by second interconnect 235.

**[0030]** **Figure 3** illustrates an example of a system with multiple processors. Processor 305 may include first processor 310 and second processor 315, which share execution resources 330, cache 335, and system bus 340. Architecture state registers 320 and 325 may hold unique architecture states of processors 320 and 325 respectively. System bus 340 couples processor 305 to controller hub 345. Controller hub 345 may be coupled to storage medium 355 by a second bus 350. Storage medium 355 may be any device that stores data. For example, storage medium may be system memory. System memory may include synchronous random access memory (SRAM), dynamic random access memory (DRAM), synchronous dynamic random access memory (SDRAM), double data-rate RAM (DDR), Rambus (R), or any other type of system memory. Storage medium 355 may also be a register, flash memory, or another level of cache 335.

**[0031]** Looking at **Figures 4 and 5**, illustrative examples for allocation of system management mode (SMM) memory space for a first processor and for a second processor within storage medium 355 are depicted. **Figure 4** illustrates a first SMM memory space 405, which may be assigned to the first processor, and a second SMM memory space 410, which may be assigned to the second processor. First SMM space 405 and second SMM space 410 overlap to create overlapping region 415.

[0032] A first non-overlapping region 420, a second non-overlapping region 425, and/or third overlapping region 435 may be used to retain separate memory spaces for each processor. For example, separate memory spaces 420 and 425 may be used to store the first and second processor's save-state area. As another example, non-overlapping region 420 may be an offset between the first and second processor's base address (SMBase), while 435 and 425 are used to store the first and second processor's save-state area. **Figure 4** also illustrates a synchronization area 430 within overlapping region 415, which may be used to store synchronization information for a first processor and/or a second processor.

[0033] Turning to **Figure 5**, an illustrative example of synchronization area 430 is depicted. Synchronization area 430 may be a synchronization byte that represents the state of a first processor and/or a second processor. As discussed later in reference to **Figures 6-10**, synchronization area 430 may be stored anywhere within storage medium 355 shown in Figure 3. Cells 505, 510, and 515 illustrate examples of values that a synchronization byte may contain to represent the states depicted in cells 520, 525, and 530 respectively. Processor states, as well as **Figures 4 and 5,** will be discussed in more detail in reference to the methods described in **Figures 6-10**.

[0034] Referring to **Figure 6**, a method for synchronizing a first and second processor before handling an SMI is illustrated by a high-level flow diagram. In block 605, a SMI is received. Often a SMI is generated to request a service from a processor. A SMI may be generated by an asynchronous (hardware) or synchronous (software) event. When a SMI is generated in the system each processor in the system should receive/latch the SMI.

[0035] As an illustrative example, the first SMI in block 605 may be generated by a controller hub 345, shown in **Figure 3**. As another example, the first SMI in block 605 may be generated by a controller (not depicted) located either within a first processor, a second processor, or separately in the system. As yet another example, the first SMI in block 605 may be generated by changing the logic level of a pin on a physical processor, such as processor 305 depicted in **Figure 3** or on a controller hub, such as controller hub 345 in **Figure 3**.

[0036] In block 610, a first processor checks the state of a second processor. Checking the state of a second processor may be done through communication between processors. As an illustrative example, **Figure 7** depicts, in block 705, how a first processor may check the state of a second processor by examining a storage medium, such as storage medium 355 in **Figure 3** and **Figure 4**.

[0037] Storage medium 355 may be any medium that stores information. As an example, storage medium 355 may be at least one register located in the first processor, the second processor, or the multiprocessor system (not depicted). As another example, storage medium 355 may be part of cache 335 or any other cache not located on processor 305. As yet another example, storage medium 355 may be system memory.

[0038] Storage medium 355 may hold direct state information of the first processor and/or second processor. Moreover, storage medium 355 may hold information representative of the first processor's and/or the second processor's state. For example, storage medium 355 may store state information in a synchronization area, such as synchronization area 430, depicted in **Figures 4 and 5**. Synchronization area 430, may store different values that represent different processor states.

[0039] As an instructive example, a first value stored in synchronization area 430 may represent that the second processor is in an inactive state. An inactive state may be any low power state, such as sleep, standby, suspend, hibernation, wait-for-SIPI, sleep, deep sleep, reset, or any other mode where the second processor does not respond to interrupts. Additionally, a second value stored in synchronization area 430 may represent that the second processor is in an active state but not in SMI mode, which is also commonly referred to as system management mode (SMM). An active state but not in SMI mode may be any state where the second processor is responding to interrupts and/or executing code, but is not in SMM. Furthermore, a third value stored in synchronization area 430 may represent that the second processor is in an active state and in SMI mode. An active state and in SMI mode, may be any state where the second processor is active and also in SMM.

**[0040]** Turning back to **Figure 5**, illustrative examples of representative values that may be stored in synchronization area 430 as a synchbyte are shown. When synchronization area 430 stores a first value 01b in cell 510, that first value represents that the second processor, is inactive/sleeping, as described in cell 520. However, when synchronization area 430 stores a second value 00b, as shown in cell 505, that second value represents that the second processor is active and not in SMI mode, so the first processor may wait for the second processor as described in cell 520. Similarly, when synchronization area 430 stores a third value 10b, as shown in cell 515, that third value represents that the second processor is active and in SMI mode. Therefore, the first processor may proceed to handle the SMI on both the first and second processors, as described in cell 530.

**[0041]** Referring again to **Figure 6**, the SMI from block 605 is handled in block 615. Often handling an SMI entails servicing the request made either by hardware or software. Handling an SMI may include anything that services the SMI request generated. For example, handling an SMI may include executing SMI handler code to service the SMI.

[0042] As shown in block 620, the first processor may handle the SMI generated in block 605 without waiting for the second processor to enter SMI mode, if synchronization area 430 stores a value, which represents that the second processor is in an inactive state. Once the second processor begins to wake-up (enter a state where it responds to interrupts), the second processor may update the storage medium to reflect its current state, such as in block 710 shown in **Figure 7**.

[0043] When entering a state where the second processor responds to interrupts and is not in SMI mode, the value in synchronization area 430 should be updated by the second processor to reflect an active but not in SMI mode state. While synchronization area 430 represents an active and not in SMI mode state, the first processor should wait for second processor to enter SMI mode before handing the SMI from block 605, as shown in block 625. Likewise, when the second processor enters SMI mode it should update the synchronization area to a value, 10b in **Figure 5**, which represents that the second processor is now in SMI mode. If the state of the second processor is active and in SMI mode, then the SMI may be handled with either the first processor or second processor on both first and second processors.

[0044] Taking a look at **Figure 8**, an illustrative example of optimized synchronization during a two processor boot sequence is shown in high-level flow diagram format. Synchronization value 805 is shown continuously through the flow diagram to illustrate what value synchronization area 430, shown in **Figure's 4 and 5**, may hold. Synchronization value 805 may be stored anywhere in storage medium 355. For this example synchronization value 805 should coincide with **Figure 5's** table of values to make the illustration simpler. Therefore, upon initialization/reset in block 810 synchronization value 805 may be set to 01b, representing that a second processor is inactive. At this time, if any SMIs are generated, the second processor may latch/receive the SMI, but should not handle them.

**[0045]** However, a first processor, after receiving an SMI, may check the state of the second processor by examining synchronization area 430 to read the synchronization value 805. The first processor may then proceed to handle the SMI without waiting for the second processor, if the synchronization value 805 is 01b, representing that the second processor is inactive. The first processor may also complete other routine boot steps, such as initializing SMI in block 815, completing power on self test (POST) in block 820, and waking the second processor in block 825.

**[0046]** When the second processor wakes-up in block 830, it should set synchronization value 805 to 00b, to represent that it is active but not in SMI mode. The first processor may enter SMI, in block 835, and should check the state of the second processor by examining synchronization value 805. Since synchronization value 805 should now be set to 00b, the first processor may wait/loop until the second processor enters SMI and sets synchronization value 805 to 10b.

**[0047]** The second processor, in block 840, may then enter SMI mode and set synchronization value 805 to 10b to represent it is active and in SMI mode. At this point, the second processor may wait/loop in SMI mode until the first processor has set synchronization value 805 to 00b to represent it is active but not in SMI mode. During the second processor's waiting, the first processor may proceed to handle the SMI in block 845 on both the first and the second processors. Once the first processor has handled the SMI on both the first and second processor, it may exit SMI mode and set synchronization value 805 to 00b in block 850. The second processor may then exit SMI mode in block 855.

**[0048]** Referring to **Figure 9**, a method for efficiently allocating system management memory space is depicted. In block 905 a first system management memory space/range, such as first memory space/range 405 in **Figure 4**, is assigned to a first processor. In block 910, a second system management memory space/range, such as second memory space/range 410, is assigned to a second processor, so that first memory space 405 and second memory space 410 overlap to create an overlapping region/range 415. The overlapping of first memory space 405 and second memory space 410 may leave a first and second non-overlapping region/range, such as non-overlapping regions/ranges 420 and 425 respectively, shown in **Figure 4**.

**[0049]** Overlapping region 415 may be used to store system management data such as SMI handler code. Overlapping region 415 may also be used to store one or both of the first and second processor's save-state area. It is readily apparent that first and second non-overlapping regions 420 and 425 may be placed in different orientation to overlapping region 415 than depicted in **Figure 4**.

**[0050]** Non-overlapping regions 420 and 425 may also be used to store any sort of data. For example, first non-overlapping region 420 may store the save-state area for a first processor, while second non-overlapping region 425 may store the save-state area for a second processor. As another example, first non-overlapping region 420 may be an offset between the first and second processor's SMM space, while save-state area 435, in overlapping region 415, and non-overlapping region 425 may be used to separately store a first and second processor's save-state area. Furthermore, non-overlapping regions 420 and 425 may be any size in memory. As an illustrative example, first and second non-overlapping regions 420 and 425 may be the size of each processor's save-state area. A typical save-state area may be 2kB but may also vary in size. The offset between first memory space 405 and second memory space 410, as shown by non-overlapping region 420 may vary in size as well. For example, the offset may be the size of the save-state area or it may be the size of the largest SMI handler code stored in either memory space.

[0051] A portion of overlapping region 415 may also store the system management state of both processors in a synchronization area 430, as shown in **Figure 4**; even though, synchronization area 430 may be placed anywhere in storage medium 355. However, it may be advantageous to store synchronization are 430 in the overlapping region, so both processors may read information from and modify synchronization area 430 easily.

[0052] As illustrated above, the need for efficient synchronization and memory allocation in system management is becoming greater as multiprocessor systems have fewer resources available to them. Allowing the processor's to communicate each others state either directly or through a storage medium, allows a multiprocessor system to efficiently synchronize, not wasting any execution time or resources. Furthermore, overlapping each processor's system management memory space/range saves valuable memory space, as well as allows the synchronization information to be readily stored and modifiable by any processor.

[0053] In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense.